CECOM

CENTER FOR SOFTWARE ENGINEERING

ADVANCED SOFTWARE TECHNOLOGY

Subject: **Final Report - Analysis of the Impact of the Ada Runtime Environment on Software Reuse**

CIN: **C02 092LA 0007**

31 MAY 1989

ANALYSIS OF THE IMPACT OF THE ADA RUNTIME ENVIRONMENT ON SOFTWARE REUSE[1]

by

Anthony Gargaro
Computer Sciences Corporation
Defense Systems Division

prepared for

U.S. Army CECOM, CSE
Fort Monmouth, New Jersey 07703-5000

December 31, 1988

Contract No. DAAB07-85-C-K524, Task 7067-34040

---

1. The material presented in this report represents the author's opinions and perceptions and should not be construed as those of Computer Sciences Corporation nor the Department of the Army.

Summary

The number of applications using the Ada language is increasing rapidly with the availability of over 200 base and derived validated compilers. Reports indicate that many of these applications have been successfully developed from reusable software parts or have been developed to yield reusable software parts. In contrast to these reports, other applications have identified deficiencies in the Ada RunTime Environment (RTE) that have necessitated restricted use of Ada with a consequent impact on the potential reuse of Ada software. Therefore, improving the reuse of software in these applications depends to some extent upon careful analysis and understanding of the issues related to software reuse and the Ada RTE.

The latter applications typically comprise embedded Mission Critical Computer Resource (MCCR) systems that must satisfy real-time performance constraints that are usually dependent upon specialized hardware. These constraints have caused the intrusion of non-Ada code into the applications and the exploitation of specific idiosyncrasies of the RTE that are within the semantic fringes of the Ada standard [DoD83]. While reducing the use of non-Ada code may not be feasible for applications that must interact with special controls, devices, and weapon systems, ameliorating the deleterious effects of deficiencies and idiomatic behavior of the RTE on the reuse of Ada parts must receive prompt attention.

Appropriate resolutions can be anticipated in the scheduled revision to the Ada standard (Ada 9X) [ABD88] if the impact of the RTE on software reuse and the attendant costs and effects are clearly identified and understood. Furthermore, it is prudent to examine resolutions outside of the scope of Ada 9X so that techniques can be devised to ensure that any remaining RTE issues are addressed. These resolutions may include standard packages to support the RTE, classes of RTEs, tailorable RTEs, and guidelines for using the RTE.

Table of Contents

Analysis of the Impact of the Ada RTE on Software Reuse

1. Introduction

There is growing evidence in some current Department of Defense (DoD) application domains that a moderate increase in programmer productivity can be realized when an aggressive policy for software reuse is employed. One important facet of this policy is the use of the Ada language combined with a methodology that adheres to disciplined design and programming practices. However, there is a concern that software reuse will be severely limited in those DoD application domains that require the extensive use of Ada constructs serviced by the Ada RunTime Environment (RTE). Unfortunately, it is in these application domains, namely embedded real-time MCCR systems, that Ada is mandated as the preferred programming language [DoD87]. Because there is a paucity of experience in using Ada for these applications, there is an immediate need to determine the validity of the aforementioned concern in terms of analyzing the impact of the Ada RTE on software reuse.

The objective of this report is to provide an initial analysis that will establish a basis on which to plan future actions. The report comprises seven sections following this introduction. Section 2 briefly reviews software reuse from the design and implementation perspectives; it deliberately provides a language-independent treatment of the subject. Section 3 relates the Ada language contribution to the design and implementation of reusable software parts. Section 4 introduces the Ada RTE. Section 5 provides an analysis of the significant constraints that RTE issues may have on software reuse. Section 6 summarizes important activities that may remove some of these constraints. Section 7 offers some specific conclusions and recommendations. Finally, Section 8 identifies documents that are referenced in the preceding sections using an organizational abbreviation of the sponsor and year of publication enclosed within squared brackets.

There are many factors that contribute to the successful reuse of software. This report addresses the construction of reusable parts and the composition of applications from reusable parts. The acquisition, distribution, management, retrieval, and use of reusable software are not addressed. In addition, it is assumed that the reader has some acquaintance with the principles of software reuse and the Ada language.

2. Software Reuse

Software reusability may be informally defined as the property of a software part to be adapted for use in the composition of applications other than the one for which it was originally developed. This definition emphasizes reuse of parts constituting an application rather than the reuse of the application itself. In addition, it specifically recognizes that in many applications it may be essential that a part be adaptable to ensure effective reusability [CSC86]. Ideally, a part may be any entity within a particular software architecture. For example, in a functionally oriented architecture, it may represent either a calling entity (client) or called entity (server).

The opportunity to reuse software is frequently proposed as one strategy for reducing the cost of developing and of enhancing the reliability of complex large-scale applications. However, software reuse usually incurs greater intellectual intensity in the initial development of a part and in many instances a decrease in its performance efficiency. For embedded real-time MCCR systems, the tradeoff between increased reliability and decreased performance may determine the degree of reusability possible.

The planned reuse of software has been practiced since the advantages for common libraries were recognized in the early days of high-level programming languages. The libraries were usually restricted to include only mathematical and statistical routines that implemented well-defined numerical algorithms. However, since then, software reuse technology has not progressed to the same level of sophistication that its counterpart technology has achieved with respect to hardware. This results from the lack of discipline and formalism used in the design and implementation of reusable software. Often, reusability is relegated to an implementation activity that is left to the discretion of the individual programmer.

While some measure of part reuse is possible through portable programming practices, it is unlikely that a part that is not specifically designed for reuse will be reused outside of a particular application. Furthermore, the application domain must be understood in order to precisely establish reusability requirements for the part. These requirements are essential for reusability to be addressed as a design criterion for the application in which the part is originally developed. This criterion must address both the reuse of existing parts and the development of new parts within an application. Unless an application is explicitly designed to take advantage of existing reusable parts, software will not be reused. Conversely, if applications are not designed to offer reusable parts, the cost to retrofit a part for reuse may eliminate any advantage to a potential application. Consequently, software reusability becomes an essential consideration of a software development methodology [SPC88].

The reuse of a software part requires a precise specification of the part's function and of its interface to client software. The interface should be sufficiently flexible and general to satisfy its scope of reuse within the application domain. The interface should be carefully designed to ensure that excessive generality of a part does not penalize performance to the extent that its reuse is compromised. Popular design principles that are commonly cited to aid in specifying functionality and interfaces for reusable parts are functional cohesion, weak coupling, and parametric abstraction [ESD86]. Parts that adhere to these principles will often achieve a high degree of composition orthogonality, which is an important criterion proposed for evaluating software reusability [CSC86, CSC87].

Complex time-critical requirements are intrinsic to embedded real-time MCCR systems. Typical applications comprise software that must be executed within rigid synchronization and time constraints that are essential for successful operation. Requirements of the software include concurrency, fault tolerance, and reliable control of resources. The specification of these requirements is extremely difficult since they frequently include dependencies on the resources available and low-level hardware/software interactions specific to the operational environment of the application. These dependencies are usually not amenable to a formal specification. There is a danger that without precise formal requirements, the resulting software may achieve successful performance using fragile algorithms that evolve from an ad hoc process of iterative behavioral refinement. Any subsequent change to the operational environment of the application may be detrimental to the correct execution of a part. The potential for software reuse is diminished because the parts are not resilient to change.

The susceptibility of parts to these dependencies increases the need for parts to be transportable. Transportability is another criterion proposed for evaluating software reusability [CSC86, CSC87]. This criterion measures the degree to which equivalent execution behavior can be achieved when an application executes in a different operational environment. Equivalent execution requires that the application performs its specified function to the extent that the new operational environment permits. Achieving identical execution requires that the application has been successfully insulated from all dependencies, explicit and implicit, on the operational environment.

Finally, the inability to adequately express time-critical requirements presents a problem that pervades many of the issues for software reuse from domain analysis to the choice of design and implementation techniques. For example, if a part is to be reused in deadline-driven applications, its execution time becomes an important attribute requiring accurate documentation with respect to the application domain. Unless a part is adequately documented its potential for reuse is severely compromised. At a minimum, the documentation must specify the function performed by the part and how the part may be reused within an application domain [CEC88a]. A corollary is that the algorithmic code of a reusable part must be readable to complete its documentation.

3. Ada and Reuse

The Ada language standard provides a significant advance towards developing reusable software parts through the use of a single high-order procedure-oriented programming language. The standard supports straightforward tenets for software development and establishes a practical basis for formalizing design and programming disciplines [CSC87]. These disciplines are essential for promoting an aggressive policy of software reuse within well-defined application domains. The tenets and associated constructs that are particularly important to software reuse are summarized in the following paragraphs.

Abstraction. Abstraction is a means of expressing essential design and implementation information at different levels of detail. The different levels of detail enable complex applications to become intellectually more manageable and understandable. When these levels of detail are consistently and systematically refined using an appropriate methodology, e.g., functional decomposition or object orientated analysis, the design and implementation yields an application comprising an infrastructure of potentially reusable abstractions. Ada directly supports both data and functional abstraction techniques through constructions such as packages, subprograms, and task types, that allow the separation of a part's functional details from its reusable interface specification.

Completeness. Completeness requires that all design and implementation detail is presented and consistently defined for an application. Ada promotes completeness through the interface specifications of packages, subprograms, and tasks, that are checked when these constructions are called or are elaborated. Consequently, the interface to a reusable part becomes a contract for its successful use that must be fulfilled by the client software.

Information Hiding. Information hiding allows the integrity of an interface to be safeguarded by restricting access to compromising implementation detail. Ada restricts such detail through limited and private types. Using these types to develop abstractions increases the reliability of an application and its constituent parts. In addition, the control of implementation detail increases the opportunity for reusing the design of a part.

Locality. Locality minimizes referential complexity in order to increase the maintainability and understanding of an abstraction. If entities referenced by a part implementing an abstraction are dispersed, logically and physically, the part is unlikely to be reusable and is indicative of a flawed abstraction. The package construction promotes locality while allowing control of remote references through the use of context clauses.

Modularity. Modularity promotes the partitioning of an application into a system structure more adaptive to design and implementation techniques. The partitioning usually results in more configurable and more manageable units for implementation. Modularity is an essential principle for software reuse since modules, or parts, rather than entire applications are reused. Units of modularity in Ada are subprograms, packages, tasks, and generic units.

Verifiability. Verifiability requires that the design and implementation of an application can be proven consistent without a demonstration of correct execution. Usually correct execution of software is established by exhaustive testing within a particular application. Therefore, for software that is to be reused in different applications, formal verification of its correctness for the domain of application is necessary. While verifiability of reusable software is currently impractical, the inherent consistency within Ada provides a basis for limited proofs of correctness of parts that have minimal dynamic operations and are serially executed.

4. Ada and RTE

A principal objective of the Ada language is to remove the dependency of embedded real-time MCCR systems software on specialized executive services. Traditionally these systems have relied on application specific executives to support the particular application programming language. This has seriously impeded the reuse of software among different applications since these executives were designed using incompatible models of the execution environment and were not readily transportable.

Ada presents to all applications a single semantic model of the execution environment. This model is commonly termed the Ada RunTime Environment (RTE) and while its actual implementation may vary significantly over different target computers, its semantics are bounded by the Ada standard. Within an Ada program the RTE is responsible for achieving concurrency, calling units, converting data, managing storage, performing I/O, and synchronizing actions. Therefore, developing parts adhering to this model not only eliminates the need for typical executive services but promotes software reuse by increasing the transportability of applications. Furthermore, parts may be developed so that their logical structure is not fractured by an inconsistent level of abstraction between the constructs of the language and those of the executive.

For example, in many instances the abstraction level of executive services are oriented to a low-level procedural interface requiring artificial abstractions to raise it to the level of the application's programming language.

Conceptually, the code executing in the target computers should comprise the application code generated by the Ada compiler, the Ada RTE, and a modest executive kernel to implement privileged low-level target operations, e.g., context switching among concurrent tasks. Together the software that supports the application code generated by the Ada compiler and the target computer may be termed an Ada Virtual Machine (AVM). Optimally, within an application domain, the AVM provides an efficient and sufficient foundation for all Ada software. Since there is reduced need and opportunity for the parts to access functions beyond those made available by the AVM, the parts from different applications should be more reusable.

5. RTE and Reuse Analysis

The previous two sections have summarized the advantages that Ada and its RTE contribute to software reuse. In this section, RTE issues that may significantly offset these advantages and handicap software reuse are presented. These issues are dominated by the following questions:

1. Is the performance of RTE implementations comparable to that of contemporary real-time executives?
2. Is the level of abstraction to the RTE appropriate for developing embedded real-time MCCR systems?
3. Is the RTE interface sufficiently defined to eschew implementation dependencies?

To an extent the questions are not mutually exclusive; for example, performance may be impacted by the level of abstraction. Therefore, similar concerns may permeate the individual analyses of these issues.

5.1 RTE Performance

An underlying assumption of using Ada for developing embedded real-time MCCR systems is that the performance efficiency of the RTE implementations are comparable to the specialized services of contemporary real-time executives designed for the same target computer configurations. Too frequently, the Ada analogues and paradigms that replace these specialized services are offered with insufficient evidence that satisfactory performance efficiency can be achieved. Unless comparable performance efficiency is possible, it is likely that vestiges of specialized executive services will continue to exist within the AVM. This will complicate software reuse since to reuse a part in a different target execution environment, the part must be purged of these executive services and retrofitted to use the Ada RTE or other executive services. The probability that this exercise will result in reliable software decreases each time the retrofit is undertaken.

While no formal evaluation criteria of RTE performance is available to produce comparative data, there appears to be consensus in the embedded real-time community that many RTEs are less efficient than contemporary real-time executives [ACM87a, ACM88b]. RTE implementations are frequently criticized for using an excessive amount of storage and for exacting high execution time penalties. These inefficiencies are sometimes exacerbated by implementation strategies within the RTE that are totally devoid of any optimization techniques. Paradoxically, it would seem that RTE implementations are frequently designed for reuse in the sense that they can be easily adapted to support the rapid implementation of different AVMs. These AVMs satisfy the Ada validation requirements but are often encumbered with general-purpose operating systems that are the antithesis of the compact executive kernel.

The use of a general purpose operating system within the AVM presents a dual threat to software reuse. The first is the imposition of significant overhead on the RTE. In some instances, restrictions imposed by the operating system upon the RTE result in execution time behavior that cripples the use of Ada. A typical example occurs when the operating system suspends the execution of the RTE to complete a service request, causing all tasks within the Ada program to become blocked. Under these conditions, applications will not use certain Ada abstractions. The second is that operating system services are usually readily accessible to the Ada application through the pragma INTERFACE. The use of this pragma may offer an inviting functional, more efficient, alternative that will undermine the use of Ada. Once such services penetrate the application, the semantic boundary of the standard is inevitably compromised.

Finally, there is an expected loss of efficiency because the RTE model is orientated towards supporting static compiler-checked abstractions. These abstractions are not adaptable dynamically to the availability or type of processing resources. From the perspective of software reuse, the loss illustrates the difficulty of the tradeoff between reliability and performance that must be accommodated within any programming language.

5.2 Levels of Abstraction

For a software part to be reusable it must execute correctly within the requirements of its domain of application. When this domain of application includes the requirements typical of embedded real-time MCCR systems, a part is often expected to satisfy "hard" timing constraints that are fundamental to the successful operation of the application. Therefore, achieving correct execution demands that the timing behavior of a part be predictable and reliable. The transformation of timing behavior into formal resource utilization abstractions becomes a central design objective of the application.

The abstractions supported by the RTE that pertain to resource utilization have been criticized by the real-time Ada community as being deficient. Generally, the critics have argued, with varying degrees of intensity, that the level of the abstractions are not responsive to the language requirements as specified in the Steelman document [DoD78]. Many abstractions are considered to be defined at an inappropriate level of detail and fail to provide sufficient control through their interface and associated functionality. In particular, the abstractions for concurrency, event control, storage management, and timing have been identified as specific issues for real-time systems [CEC88b].

5.2.1 Concurrency

The Ada concurrency model supports asymmetric communication and synchronization among parallel separate threads of control within a single Ada program. The threads of control are represented as autonomous processes called tasks. The mode of parallelism is multiple instruction multiple data (MIMD), and the model provides a unified abstraction for expressing the classical paradigms of protecting shared data and of message (signal) processing. Furthermore, the processing capacity of the execution environment is transparent to the application, thereby promoting the notion that an Ada program may be executed with minimal regard to whether the target computer is a single, multiple, or distributed processor. Unfortunately, the conceptual elegance and capacity transparency properties are not universally perceived as benefits for achieving "hard" real-time deadlines.

There are several reasons for concern regarding the abstractions of the Ada tasking model. The advantage of integrating concurrent execution into the programming language is accompanied by a penalty of increased semantic complexity in the language. This is evident in fundamental operations such as task activation termination, and abortion that require significant RTE support and are vulnerable to misuse. The level of abstraction provided by the rendezvous semantics, while adequate for pedagogical examples of concurrency, may be quite unsuitable for embedded real-time MCCR systems. For example, the RTE must be carefully instrumented and restricted to support formal task-scheduling algorithms, and arcane rendezvous paradigms are necessary to mimic simple prevailing models of periodic task execution. This leads to potential dependencies upon the RTE and excessive execution-time overhead. Furthermore, rudimentary concurrent programming abstractions, e.g., buffers, monitors, semaphores, often require the introduction of intermediary (agent) tasks. Their impact on software reuse can be detrimental to performance and reliability.

While comparative arguments in favor of the rendezvous model are convincing [Hon86], they fail to recognize the inherent asynchronism of embedded real-time MCCR systems. Consequently, reusable parts that must accommodate asynchronism within the application may be unexpectedly thwarted by the enforced synchronization of the RTE.

The degree to which the model can retain the capacity transparency of the target execution environment is problematic. The development of reusable parts without regard to whether the target execution environment is a single processor or a configuration of loosely coupled computers does not appear to be practical. Once the possibility of distributing the execution of an Ada program is raised, the issue of program partitioning becomes significant. Partitioning strategies may be described as either "post-partitioning" or "pre-partitioning". Each strategy requires RTE support that may affect software reuse when, for example, objects are located such that referencing them incurs an unacceptable performance efficiency overhead. One rudimentary scheme proposed for developing reusable parts for distributed execution environments using pre-partitioning requires adherence to specific guidelines for composing reusable applications [CEC88c]. However, the scheme assumes the cooperation of the RTE to support remote references across loosely coupled computers.

5.2.2 Event Control

The control of events is a fundamental requirement of embedded real-time MCCR systems. Events may be synchronous or asynchronous. The primary support for controlling events in Ada is through a facility for the RTE to explicitly bind a hardware interrupt to a predetermined thread of control, i.e., an entry of a task. In addition, the RTE may elect to implicitly bind an interrupt through a predefined exception. A less well-defined abstraction is provided through the standard package for low-level input and output operations. None of these abstractions are defined at a level that ensures any degree of reuse for parts that utilize them.

The explicit binding of interrupts to task entries requires the interrupt to be named using a target dependent identification, i.e., SYSTEM.Address. Interrupts cannot be disabled, and the disposition of unserviced interrupts is undefined. Therefore, dynamically changing the processing associated with an interrupt cannot be achieved safely. Furthermore, RTE implementations are frequently required to specify compiler-dependent restrictions in order to provide guaranteed interrupt service within a prescribed time.

The implicit binding of interrupts to predefined exceptions is analogous to a system trap. Due to the synchronous property of exceptions, such exceptions cannot result in the direct execution of a separate thread of control, viz., a different task. In addition, their contribution as abstractions for reusable software is faulted because they are necessarily compiler dependent.

The control of external events through low-level input and output operations provides a reusable interface to the RTE but with functionality that is essentially dependent upon the target device for the operations. To monitor the occurrence of an event requires that each event be assigned a separate task to resume execution after the input operation, i.e., Receive_Control, has completed. The response time to an event will be difficult to predict because of the potential delay in completing the operation.

5.2.3 Storage Management

The management of dynamic storage by the RTE is primarily effected through Ada access types. Additional control is provided through generic subprograms, pragmas, and representation clauses. The motivation for the abstractions is reliability rather than performance efficiency [Hon86]. While reliability is a necessary attribute for reusable software, the potential loss of efficiency and functionality offered by the abstractions to the RTE may restrict their use for embedded real-time MCCR systems. Traditionally, the target computers for these applications have imposed stringent limitations on storage capacity requiring that the resource be economically utilized. At a minimum, the allocation and deallocation of storage must be flexible and efficient.

The RTE allocates storage explicitly through allocators. Allocators allow the dynamic creation of a variety of logical data structures that are conducive to reusable software. An omission is the ability for a part to exercise control of the mapping of logical data to storage through the RTE. For example, when different classes of storage are available, it is often desirable to allocate specific structures to a particular class, e.g., real/virtual, protected/unprotected, to increase part efficiency and reliability. In addition, there are instances when it is important that this control be available when the RTE implicitly allocates storage during program execution. For example, when two tasks are executed with predefined periodicity, allowing one task to overlay the other provides practical storage control without compromising part reuse.

The explicit deallocation of storage by the RTE may only be influenced through the pragma CONTROLLED for access types and the generic subprogram UNCHECKED_DEALLOCATION for access objects. As the name implies, the latter facility impacts reliability by introducing the possibility of anonymous (dangling) access values. Unfortunately, without a well-defined storage reclamation strategy provided by the RTE, the subprogram may be used extensively by embedded real-time MCCR systems. While the pragma allows the RTE to inhibit automatic storage deallocation, there is no comparable interface to ensure that storage is reclaimed for an access type collection. Storage reclamation becomes an RTE dependency of significant impact by causing unpredictable application execution when reusing parts that have both space and time constraints.

5.2.4 Timing

Timing considerations have infiltrated many of the previous discussions because of their dominant role in embedded real-time MCCR systems. Therefore, the impact of the RTE with respect to time on software reuse is a significant issue since the Ada abstractions for time depend upon the RTE implementation. Unfortunately, the Ada timing abstractions have been recognized as lacking both a reliable interface and essential functionality [IDA88]. This leads to difficulty in developing reusable parts that require precise control over timing, in particular, execution timing.

The abstractions for timing should support the bounding of the elapsed time between events, and the execution time of sections, including separate control threads, of code. The former is often necessary to confirm functional requirements while the latter is required to facilitate resource scheduling and to detect aberrant execution. The language timing abstractions to the RTE, the Ada delay statement and Calendar package, do not adequately address these requirements. For example, using these abstractions as part of the tasking model to simulate a simple reusable cyclic scheduler is not straightforward because of the unguaranteed accuracy of the delay and uncontrolled latency in timing computations. Additional imprecision is introduced into timing computations by the absence of attributes that would extend the model for numeric accuracy to time, e.g., formalizing a relationship between SYSTEM.Tick and Duration'Small. This prevents reusable parts from determining the timing characteristics of the target execution environment and providing for appropriate contingent action when necessary.

Finally, the timing abstractions fail to separate functionality by distinguishing between standard clocks and real-time clocks, and do not support a reasonable model of time for distributed target execution environments.

5.3 RTE Dependencies

Abstracting the traditional functions of specialized executives into RTE constructs would seem to guarantee predictability and reliability of reusing an Ada part over its application domain. Unfortunately, Ada constructs having implied temporal semantics are specified at an imprecise level of abstraction and result in restricted functionality or permissive semantic specifications. The level of abstraction compromises uniformity of implementation among RTEs resulting in dependencies upon the RTE. Since the dependencies are not specified by the application domain, this leads to software that is less reusable.

These dependencies result in unpredictable or unreliable timing behavior of a part when reused outside of the application in which it was originally developed. This condition is further exacerbated by the informality in the requirements specification of application domains for embedded real-time MCCR systems. The documentation for the part will most likely propagate this informality in critical areas of timing behavior, thereby camouflaging the part's lack of predictability and reliability.

A further problem is caused by the existence of subtle implicit dependencies upon the RTE. This problem is manifest when combining reusable parts in an execution environment that is different from those used to originally develop the parts. For example, when combining two parts, three different RTEs must be considered; two in the original execution environments, and one in the new execution environment. This can precipitate a particularly insidious problem because the applications enclosing the parts may have been successfully transported to the new execution environment and have demonstrated functionally identical execution to that achieved in their original execution environments. Furthermore, the parts may have been separately combined with other parts and have achieved successful execution, thereby giving a high degree of confidence that the parts are free of RTE dependencies. It is only when combining the parts dependent upon conflicting RTE behavior that failure or aberrant execution will occur [CSC86].

6. RTE/Reuse Initiatives

The impact of the Ada RTE on software reuse, in many instances, raises issues that have been the source of technical debates within the community of real-time Ada programmers for some time. These debates have resulted in the formation of several ongoing initiatives that may ameliorate many issues which in turn will improve the opportunity to develop reusable software. Three directly relevant initiatives are the preparation of an Ada Reusability Handbook, the specification of a Catalog of Interface Features and Options, and the development of a Model RunTime System Interface.

6.1 Ada Reusability Handbook

The Ada Reusability Handbook (ARH) has been developed by Computer Sciences Corporation in conjunction with the U.S. Army CECOM, CSE [CSC87, CEC88c]. The purpose of the ARH is to identify recommended guidelines for writing reusable parts. The guidelines are accompanied by a preliminary annex devoted to a discussion of writing reusable parts in the presence of real-time requirements.

The ARH is oriented towards effective reusability, i.e., developing parts that have a high pragmatic potential for reuse. The guidelines emphasize that through the use of defensive programming techniques and a thorough understanding of Ada constructions supporting software reuse, effective reusability can be achieved within the constraints of the design. To increase the understanding for a guideline, each guideline is categorized with respect to one of four criteria defined to evaluate software reusability. It is expected that the ARH will eventually provide a modest contribution to formalizing a coding discipline for software reuse. Such a discipline would increase the reusability of part interfaces while protecting their functionality from idiomatic RTE behavior.

6.2 Catalog of Interface Features and Options

The Ada RunTime Environment Working Group (ARTEWG) under the sponsorship of the Association for Computing Machinery's (ACM) Special Interest Group on Ada (SIGAda) has specified a preliminary Catalog of Interface Features and Options (CIFO) for the Ada RTE [ACM87b]. The purpose of the CIFO is to provide common user-RTE interfaces to RTE capabilities that were intentionally (necessarily) omitted from the Ada standard.

The CIFO may be viewed as a consistent and systematic extension to the Ada Virtual Machine (AVM) as described earlier. Furthermore, the rationale for the CIFO states:

> "Common interfaces are clearly needed to make the development of high-quality software practical for the full range of applications that Ada was intended to serve, especially the domain of embedded real-time systems. This Catalog of Interface Features and Options is being developed to promote commonality with respect to such implementation dependencies, and to promote reusability and transportability."

The current CIFO does not address all of the potential issues. However, the issues that are addressed respond to the three questions posed in Section 5. Careful attention is given in the catalog entries that have been specified to improving RTE performance, refining RTE abstractions for real-time requirements, and reducing RTE dependencies. For example, an interface is specified that guarantees the execution of time-critical sections of code to be completed without preemption. This exemplifies a refined level of abstraction that will improve performance and, by presenting a common interface specification, will remove an RTE dependency.

6.3 Model RunTime System Interface

The Ada RunTime Environment Working Group (ARTEWG) under the sponsorship of the Association for Computing Machinery's (ACM) Special Interest Group on Ada (SIGAda) has specified a preliminary Model RunTime System Interface (MRTSI) for Ada [ACM88a]. The purpose of the MRTSI is to describe an interface to the "executive" functions of Ada, e.g., tasking, in order that these functions can be implemented and tailored for real-time embedded systems.

The MRTSI not only promotes software reuse by facilitating the tailoring of Ada RTEs to application domain requirements, it introduces into the domain of compiler technology the potential reuse of compiler parts, viz., those associated with implementing the Ada runtime system. In addition, the increased visibility into the Ada runtime is an incentive for compiler vendors to reduce RTE dependencies through improved cooperation. Finally, future revisions of the MRTSI are expected to support the CIFO, thereby enhancing the contribution of each initiative toward software reuse.

7. Conclusions and Recommendations

The expected potential for software reuse within embedded real-time MCCR systems is difficult to predict. However, it is evident that Ada will be the primary influence on determining this potential in the near term. The previous sections have amplified both the strengths and weaknesses of using Ada to develop reusable parts. It has been observed that the principal threats to software reuse are the impediments that the RTE presents to developing applications with "hard" real-time requirements. In many instances, the impediments result from language design decisions and tradeoffs that need to be reviewed with the experience that has been gained from compiler implementations and applications development [ABD88].

The revision to the Ada standard is the preferred choice for addressing these impediments. Some revisions are relatively straightforward and are supported by real-time Ada programmers, e.g., consistent specification of priority, while others are less straightforward and schismatic, e.g., asynchronous exceptions. A potential conflict in revising the standard is that in some instances, the emphasis is on minimizing the semantic freedom for the RTE, while in other instances, the emphasis is on increasing its freedom. Compounding the dilemma are proposals for introducing "light-weight" tasks and support for distributed partitions. Clearly, it is impractical to envisage a harmonious reconciliation of these proposals within Ada 9X, particularly with the objective of increasing software reuse. Therefore, it is recommended that the revisions in Ada 9X be achieved within the existing semantic boundaries of the current standard. The foci should be limited to removing inconsistencies and providing specifiable semantic freedom for the RTE that does not compromise the existing standard, e.g., allowing entry queues to be serviced in an order other than the default of FIFO.

Several other recommendations become necessary given the one regarding Ada 9X. They recognize the value of the ongoing initiatives identified in Section 6 as reasonable approaches to resolving issues that are beyond the province of Ada 9X. Therefore, the recommendations are to continue these initiatives with increased direction and funding for the purpose of improving the reuse of software parts for embedded real-time MCCR systems. The requirements for software reuse are sufficiently demanding that other requirements, viz., performance, are included. In addition, it seems appropriate to establish initiatives for developing architectural classes of AVMs and to investigate domains of application for the formal qualification of reusable parts.

8. References

[ACM87a] - International Real-Time Ada Issues Workshop. ACM SIGAda Ada Letters, Volume VII Number 6, 1987.

[ACM87b] - Catalog of Interface Features and Options. ACM SIGAda RunTime Environment Working Group, December 1987.

[ACM88a] - A Model RunTime System Interface for Ada. ACM SIGAda Ada RunTime Environment Working Group, August 1988.

[ACM88b] - International Real-Time Ada Issues Workshop. ACM SIGAda Ada Letters, Volume VIII Number 7, 1988.

[ABD88] - Ada Board's Recommended Ada 9X Strategy. Ada Board, September 1988.

[CEC88a] - Real-Time Technical Interchange Meeting : Real-Time & Reuse Working Group. U.S. Army CECOM, CSE, July 1988.

[CEC88b] - Issues Involved in Developing Real-Time Ada Systems. U.S. Army CECOM, CSE, July 1988.

[CEC88c] - Real-Time Requirements Annex : Ada Reusability Handbook. U.S. Army CECOM, CSE, 1988 (To be published).

[CEC88d] - Ada Reusability Handbook. U.S. Army CECOM, CSE, 1988 (To be published).

[CSC86] - Ada Reusability Study. Computer Sciences Corporation, Technical Report SP-IRD 9, August 1986.

[DoD78] - Department of Defense Requirements for High-Order Computer Programming Languages - Steelman. U.S. Department of Defense, 1978.

[DoD83] - Reference Manual for the Ada Programming Language - ANSI/MIL-STD-1815A. U.S. Department of Defense, February 1983.

[DoD87] - Computer Programming Language Policy. U.S. Department of Defense Directive No. 3405.1, April 1987.

[ESD86] - Program Office Guide to Ada, Edition 2. Electronic Systems Division, Air Force Systems Command, ESD-TR-86-282, October 1986.

[Hon86] - Rationale for the Design of the Ada Programming Language. Honeywell SRC & Alsys, Inc., 1986.

[IDA88] - Workshop on Real-Time Systems & Ada : Ada Time Abstractions Working Group. Institute for Defense Analyses, June 1988.

[SPC88] - Guidelines for Designing Reusable Software. Software Productivity Consortium, May 1988.